Automatic Compilation of Objects to Counters in Automatic Planning. Case of study: Creation Planning

Technical Report UC3M. URI http://hdl.handle.net/10016/19707

Tomás de la Rosa and **Raquel Fuentetaja**
Departamento de Informática, Universidad Carlos III de Madrid
Avda. de la Universidad, 30. Leganés (Madrid). Spain
rfuentet@inf.uc3m.es, trosa@inf.uc3m.es

Abstract. In classical planning, all objects should be represented as constants explicitly, even though their names could be irrelevant, which produces severe instantiation problems. This is specially problematic in tasks with actions for creating new objects, as it involves to estimate how many potential new objects will be necessary to solve the task. We propose a new automatic compilation from the classical to a numeric planning model to represent objects with irrelevant names using numerical functions. The compilation reduces the size of the instantiation and avoids the need of estimating the number of future objects in advance. The compiled planning task can be solved several orders of magnitude faster than its equivalent classical model.

1 INTRODUCTION

In the last years, most of the research in Automated Planning has focused on building efficient planning engines. The main tendency has been to develop techniques for improving planning algorithms. However, the study of techniques to generate efficient representations has received much less attention. In this paper, we propose an automatic translation to represent planning domains efficiently for a particular type of planning tasks. This translation modifies the representation of objects with irrelevant names. The detection of irrelevances as well as the design of techniques to deal with them efficiently are crucial points to build scalable planners [9, 5].

Current models for Automated Planning assume all objects exist at the beginning of the planning task. In fact, in PDDL, the standard Planning Domain Definition Language, all objects should be defined as constants. This is not so natural for domains containing actions for creating new objects (*creation planning* tasks) as they do not exist initially. Consider a domain with actions to divide objects. For instance, an action splitting a single piece into several smaller pieces. Then, all possible future pieces in the environment have to be defined in advance, even though its number is not known initially. Estimating this number is not necessarily an easy task. Moreover, its computation can be considered as part of solving the problem.

Creation planning tasks present two main problems: (1) the representation with constants is inefficient when the names of non-existing objects are irrelevant for the planning task; and (2) the number of these objects should be estimated *a-priori*. There are two options to solve automatically (2): to build a domain-dependent external reasoner able of computing in advance the number of constants required to solve the task, or to include an arbitrary large number of symbols to represent them. The latter has the bad consequence of degrading the planner performance since the instantiation grows exponentially with the number of constants.

In this paper we study the problem of *creation planning* from different perspectives. Then, we propose a new automatic compilation of the planning task into a different representation in which the properties of some objects are represented as numerical functions. The compilation mitigates the problem of defining *a-priori* all constants in creation domains. However, it can only be applied when these objects have irrelevant names. This paper contributes with the identification and treatment of this type of irrelevance.

2 MODELLING OBJECT CREATION

When new created objects are resources that their object states remain the same overall the planning process, they can be just counted (i.e., its quantity is represented by a numerical state variable) rather than modelled as individual objects. Nevertheless, when the domain actions perform changes over them, it is more natural to represent these objects with symbols, in order to recognize each object and the features of its particular state.

Regarding the symbolic modelling, there are two basic options to represent a simulated scenario where objects are created: a *pool model* and a *control model*. In the *pool model*, all symbols for representing potential new objects are considered equal. Actions creating objects select them arbitrarily from a pool of free symbols. The domain contains a predicate to indicate a symbol has not been used yet, e.g., *(notexist ?obj - type)*. Each action creating new objects includes this predicate as a precondition. Then, its effects delete the *notexist* predicate, and add some predicates to characterize the new object. The problem file should contain a sufficient number of defined symbols of the corresponding type. Also, the initial state should include as many *(notexist object_i)* as the number of defined free symbols.

In the *control model*, symbols for new objects are selected in a strict order which is controlled in action preconditions. Only the next symbol in a stack of free symbols can be selected.[1] The number of instantiated actions is the same as in the pool model. However only one instance of each creating action is applicable in a state. This can be modelled using a predicate to represent the order e.g., *(next ?ob-*

[1] For the sake of simplicity we consider all objects being created are of the same type, though a generalization for several types is straightforward.

ject1 - type ?object2 - type), and a *pointer* predicate, e.g., *(current ?object - type)* representing the last generated object.

For both representations the number of future objects should be decided in advance, which is a problem: defining an arbitrary large number of symbols leads to an overwhelming combinatorial explosion of grounding, but defining the right number of symbols could be a difficult task that needs to be done outside of the planning process.

An alternative would be the modification of the classical planning model to handle additional objects not enumerated in the problem definition. However, from the first-order logic perspective this means that the universe of discourse is not fixed, or at least it is not specified explicitly at the beginning of the reasoning. From the point of view of the representation language it would be straightforward to include a new directive for declaring new objects in action effects. Nevertheless, with this new model the planning task would be undecidable in the general case, since the state space can be infinite. With an infinite state space, no algorithm can prove a problem is unsolvable. Instead of creating a new planning model we explore a different representation for alleviating the problems of the classical model.

2.1 CREATION DOMAIN EXAMPLE

To illustrate our compilation we will use the pizza domain. In this domain a robot waiter can cut a whole pizza or any slice into two new slices of equal size. The robot arm can also serve any slice to a guest. Initial states have one or several slices placed on one or several trays. Goals consist of having all guest served with slices of equal size.

A pizza slice has a size, *(pizzasize ?x - slice ?z - size)* and is either on a tray, *(ontray ?x - slice ?y - tray)* or holding from a tray, *(holding ?x - slice ?y - tray)*. There are actions to *hold* a slice from a tray, to *leave* a slice on a tray, to *cut* a slice in two equal-size slices, to *first-serve* a slice to a person while deciding the size for all guests and to *serve* the rest of slices. Served slices are neither in a tray nor holding. Figure 1 depicts the pool model action for cutting a pizza in two parts. Semantically, the objects represented by the constants instantiating *?s1* and *?s2* are new, since these constants are unused symbols before the action application.[2]

```
(:action cut
:parameters (?slice ?s1 ?s2 - slice ?t - tray ?z ?zhalf - size)
:precondition (and (not (= ?s1 ?s2))
        (holding ?slice ?t) (pizzasize ?slice ?z)
        (nextsize ?z ?zhalf)
        (notexist ?s1) (notexist ?s2))
:effect (and
        (freearms) (not (holding ?slice ?t))
        (not (notexist ?s1)) (not (notexist ?s2))
        (ontray ?s1 ?t) (pizzasize ?s1 ?zhalf)
        (ontray ?s2 ?t) (pizzasize ?s2 ?zhalf)))
```

Figure 1. Pool model action for cutting a pizza in two parts.

2.2 Numerical Model for Object Creation

At the moment a new object is created, the name itself is not relevant. In fact, the symbol assigned to it is interchangeable between the rest of available free symbols. Then, it makes no sense for a search algorithm to branch on different names, given that permutations of available symbols leads to automorphisms of the search tree. Depending on the relations between a new object with other objects, the *name irrelevancy* may be true throughout the planning process. In such cases, there exists a possible numerical representation that a domain modeller may not visualize as a first choice. Our compilation automatically generates this numerical model from a pool model provided by the domain modeller.

In the generated numerical representation, the properties/relations relevant to created objects are represented using numerical functions instead of logic facts. These numerical functions are unbounded counters of the number of objects with one or more properties/relations. This representation has important advantages: (1) it avoids the need of deciding the right number of future objects in advance; (2) it avoids the search algorithm to branch on different irrelevant object names; and (3) it can reduce the grounding size.

Defining the right number of future objects may be more complicated than it seems, as it should include the creation of intermediate objects. Consider a problem in the pizza domain with two pizzas, each one on a tray, and eight guests to serve. How many *slice* symbols should be declared?[3]

The main drawback of the representation through unbounded counters is that it may lead to an infinite state space. However, when the properties/relations are finite, the state space remains also finite. Specifically, in the pizza domain, sizes of slices are finite which makes impossible to generate an infinite number of slices.

3 RELATED WORK

Created objects can be interpreted as resources produced during planning. The notion of resources here is very general. We consider as resource any object not involved in goals that can be used to achieve them. Previous work has proposed to represent resources with irrelevant identities using numeric functions [4]. The direct benefit of this functional representation is a reduction in the number of instantiated actions. However, this decision about representation is left to the domain designer and therefore it relies on his/her expertise. In contrast, we propose an automatic translation able of recognizing object types with irrelevant names and defining the corresponding functions to encode all their properties.

Our compilation solves the problem of deciding the number of symbols for created objects. It also handles the instantiation explosion due to object symmetries, previously studied in a more general scope [3, 10], but exploited at search time. In the numeric model, symmetries w.r.t. created objects do not exist, which means we are solving the problem before searching.

The numeric representation is related to general Petri Nets. For STRIPS planning problems there is a direct translation to 1-bounded Petri Nets [7, 1], where places can have at most one token and arcs have a cardinality of one. However, in general Petri Nets places represent unbounded counters. Given a 1-bounded Petri Net, under several conditions, an equivalent general Petri Net can be generated just by joining the output places representing the same properties/relations for several objects and increasing the cardinality of the corresponding arc. In the same way, a pool model, under several conditions, can be automatically compiled into a numeric representation. We identify these conditions and provide a method for the automatic compilation.

[2] This action can be also modelled by re-using the symbol *?slice*, but we want to show the general case in which an action can create several objects.

[3] At least 14 slices, to serve all guests with pizza quarters. Surprisingly, this is the largest pizza problem that state-of-the-art planners can solve.

4 TRANSLATION TO THE NUMERIC MODEL

We consider typed STRIPS planning tasks $\Pi = (\mathcal{T}, \mathcal{C}, \mathcal{P}, \mathcal{A}, \mathcal{I}, \mathcal{G})$, with negated equality.[4] In Π, $\mathcal{T}$ is a set of types, $\mathcal{C}$ a set of typed constants for representing objects and $\mathcal{P}$ a set of predicates. $\mathcal{A}$ is the set of actions which $pre(a)$, $add(a)$ and $del(a)$ are the action preconditions, positive and negative effects respectively. $\mathcal{I}$ is the initial state and $\mathcal{G}$ the goals. A plan π is an action sequence $\langle a_1, \ldots, a_n \rangle$ that when applied to $\mathcal{I}$ reaches a state where facts in G are true.

We denote the subset of objects of a particular type $t \in \mathcal{T}$ as $\mathcal{C}_t$, and the subset of predicates involving objects of type $t \in \mathcal{T}$ as $\mathcal{P}_t$. For the sake of simplicity, we define the compilation for domains where predicates $p \in \mathcal{P}_t$ have a maximum arity of two. Besides, we assume a plain type hierarchy. [5]

The compilation is designed for planning tasks in which objects of certain type have irrelevant names. Intuitively, given task Π, the objects of type $t \in \mathcal{T}$ have irrelevant names when: (1) they do not appear in the problem goals; and (2) all predicates defined for them are just to characterize them. For instance, the color of a block or the location of a truck are predicates to characterize blocks and trucks. Formally:

Definition 1 (Irrelevant name condition). The objects $c \in \mathcal{C}_t$ have irrelevant names for task Π if the following property is a state invariant: every predicate $p \in \mathcal{P}_t$ is either unary or it represents a *partial function*, $\forall x_1, x_2 \in \mathcal{C}_{t'}$ if cR_px_1 and cR_px_2 then $x_1 = x_2$.

A planning task holding the irrelevant name condition for type t is denoted by $\Pi_{\hat{t}}$. Partial functions w.r.t type t restrict predicates to represent properties for characterizing objects of type t. Given that object identities of type t are not necessary in $\Pi_{\hat{t}}$, for each state is sufficient to identify the number of objects with a particular combination of properties. Therefore, it is possible an alternative representation comprising a set of counters, one for each combination of properties.

Definition 2 (λ-fluent or counter fluent). A λ-fluent f of t is a numerical fluent that counts the number of objects in $\mathcal{C}_t$ affected by the same combination of predicates in a state s. λ-fluents are named with the concatenation of all predicates names in the combination plus t. The set of facts associated to a λ-fluent is denoted as $atoms(f)$.

The lifted representation of a λ-fluent is a PDDL *function*. The arguments are the set of variables in the corresponding predicate combination, but omitting the variables of the type t. For instance, *(ontray_pizzasize_slice ?y - tray ?z - size)* counts the number of slices of a specific size on a certain tray.

The number of possible λ-fluents is exponential in $\mathcal{P}_t$, but in practice not all properties are stated at the same time. We exploit mutex information to generate a reasonable small number of λ-fluents. In mutex groups, only one atom can characterize an object at the same time. Specifically, we use the SAS+ translation of the planning task [6], but only considering variables obtained from mutex groups with parameters of type t and reachable facts involving predicates with arguments of type t not included in these mutex groups. Reachable facts not belonging to a mutex group are considered as independent groups of just one element. Thus, the space of combinations of properties is the Cartesian product of these groups. For instance, given the mutex groups of type *slice* in the pizza domain, $G_1(slice) : [holding(s,t), ontray(s,t), \textit{none-of-those}]$ and $G_2(slice) : [pizzasize(s,z), notexist(s)]$, the resulting Cartesian product is:[6]

$$\begin{aligned} G_1 \times G_2(slice) \quad = \{ \quad & (holding(s,t), pizzasize(s,z)), \\ & (ontray(s,t), pizzasize(s,z)), \\ & (pizzasize(s,z)), \\ & (notexist(s))\} \end{aligned}$$

Every possible sub-state of an object of *slice* type can be represented by instantiating an element of this product. Thus, the Cartesian product contains all the relevant predicate combinations for generating λ-fluents. The λ-fluents obtained for the pizza domain are:

$$\begin{aligned} F_{slice} \quad = \{ \quad & (holding\_pizzasize\_slice\ ?t - tray\ ?z - size), \\ & (ontray\_pizzasize\_slice\ ?t - tray\ ?z - size), \\ & (pizzasize\_slice\ ?z - size), \\ & (notexist\_slice)\} \end{aligned}$$

The first one represents the number of slices of certain size being held from a particular tray; the second one the number of slices of a size on a tray; the third one the number of objects with a size and without any other property; and the last one the number of non-existing objects.

Definition 3. The **compilation** $\Lambda(\Pi_{\hat{t}})$ translates task $\Pi_{\hat{t}}$ into $\Pi_{\lambda(t)} = (\mathcal{T}', \mathcal{C}', \mathcal{P}', F_t, \mathcal{A}', \mathcal{I}', \mathcal{G})$ where $\mathcal{T}' = \mathcal{T} \setminus \{t\}, \mathcal{C}' = \mathcal{C} \setminus \mathcal{C}_t$, $\mathcal{P}' = \mathcal{P} \setminus \mathcal{P}_t$ and F_t is the set of relevant λ-fluents of t. $\mathcal{A}'$ and $\mathcal{I}'$ are the new set of actions and initial state respectively, both re-written to replace literals affecting t with the corresponding λ-fluent.

Now, we describe how to rewrite $\mathcal{A}$ as $\mathcal{A}'$ and $\mathcal{I}$ as $\mathcal{I}'$ in terms of λ-fluents. Let L be a set of positive literals, then, we denote as L_v the subset of literals in L over a variable or constant v. Specifically, L_v can be the subset of the preconditions, add, and delete effects of an action involving a variable v, or the subset of literals in the initial state $\mathcal{I}$ involving a specific constant.

Given an action variable (parameter) or constant, its preconditions and effects define *partial sub-states* for the objects instantiating this variable. We assume that all predicates in the delete effects appear in the preconditions. This is a reasonable restriction when dealing with state invariants [2]. In our case it allows us to recognize all transitions between object sub-states even though they are partial sub-states. On the other hand, the initial-state and λ-fluents will always refer to complete sub-states, so we have to define the unification of λ-fluents considering the different cases.

Definition 4 (Unification of λ-fluents $f \in F_t$). Let L_v be a set of literals referred to the variable or constant v. Let $\phi = \{v_i/v_j\}$ be a substitution for some variables v_i in the atoms of a subset S of λ-fluents (i.e. in $\bigcup_{f \in S} atoms(f)$), by variables or constants v_j in L_v. The unification of a λ-fluent $f \in F(t)$ is defined for the following cases:

1. Unification with action parts for action $a \in \mathcal{A}$: when $L_v = pre_v(a) \cup add_v(a) \cup del_v(a)$, we say that f unifies by substitution ϕ with
   (a) the preconditions of a over v, $pre_v(a)$ if $S = \{f\}$ and
   $$pre_v(a) \subseteq \phi(atoms(f))$$
   (b) the positive effects of a over v, $add_v(a)$, if $S = \{f', f\}$, f' unifies with $pre_v(a)$ by the same substitution ϕ, and:
   $$\phi(atoms(f')) \cup add_v(a) \setminus del_v(a) = \phi(atoms(f))$$

[4] '=' built-in predicate and its negation. Negated equality is of especial importance in creation domains that split objects.

[5] This can be assumed without loss of generality since type hierarchies can be flattened easily, generating an equivalent domain.

[6] The *none-of-those* value introduced by the algorithm to generate SAS+ variables [6] is omitted in the Cartesian product.

(c) the negative effects of a over v, $del_v(a)$, if $S = \{f\}$, f unifies with $pre_v(a)$ by the same substitution ϕ, and:

$$del_v(a) \subseteq \phi(atoms(f))$$

2. Unification with the initial state: when $L_v = \mathcal{I}_v$, we say that f unifies by substitution ϕ with $\mathcal{I}_v$ if $S = \{f\}$ and

$$\mathcal{I}_v = \phi(atoms(f))$$

Regarding preconditions, $\phi(atoms(f))$ should be a superset of $pre_v(a)$ since $atoms(f)$ defines a complete sub-state while $pre_v(a)$ defines a partial sub-state. In particular, empty preconditions unify with all λ-fluents. Thus, several variables of f can remain free after applying substitution ϕ to f (i.e. they do not appear as parameters of a). We denote as $f\phi$ the application of substitution ϕ to a λ-fluent $f \in F_t$. If $pre_v(a)$ is rewritten using $f\phi$, free variables will constitute new parameters of the resulting action.

The unification of λ-fluents with action effects depends on λ-fluents unifying with preconditions, since λ-fluents describe complete sub-states and linked variables in preconditions and effects should remain linked in λ-fluents. In other words, a λ-fluent unifying with the $pre_v(a)$ determines unique λ-fluents that unify with $add_v(a)$ and $del_v(a)$ respectively.

Regarding the initial state, we recall it defines complete sub-states for any constant, so when applying the substitution to $atoms(f)$ we exactly obtain the atoms $\mathcal{I}_v$.

Note that λ-fluents $f \in F_t$ do not contain variables of type t. Thus, $f\phi$ is not affected by the substitutions of variables of type t.

Definition 5 (Cardinality of a λ-fluent substitution $f\phi$). Let L_V be a set of literals over a set of variables/constants V. Let $V_{f\phi}^{L_V}$ be the subset of variables $v \in V$ such that λ-fluent f unifies with L_v by substitution ϕ, producing $f\phi$. Then, the cardinality of $f\phi$ in L_V is the number of elements in that set, denoted by $|V_{f\phi}^{L_V}|$.

Intuitively, the cardinality of a λ-fluent substitution $f\phi$ is the number of variables/constants of type t whose literals unify with f by the same substitution ϕ.

Translation of $\mathcal{A}$ to $\mathcal{A}'$ consists of substituting literals with the corresponding λ-fluents. An action part can be replaced by any substituted λ-fluent, $f\phi$, unifying with it, even when this part is empty. Thus, each action in $\mathcal{A}$ which preconditions are relative to partial sub-states will generate as many new actions in $\mathcal{A}'$ as different λ-fluents (complete sub-states) unify with its preconditions. Therefore, $|\mathcal{A}'|$ is always equal or higher than $|\mathcal{A}|$. However, the λ-fluents to rewrite effects are determined univocally by the λ-fluent selected to replace preconditions. Besides, it is necessary to consider that several variables of the same type can be replaced by the same $f\phi$ which increase its cardinality.

Assuming $f\phi$ is selected to replace the corresponding part of action a for a set of variables/constants V of type t, we apply the following **syntactical rules** to rewrite this action part.

- **Preconditions** involving the variables in V, $pre_V(a)$ are converted in *greater-than* preconditions using the rule:

$$pre_V(a) \text{ are replaced by } (>= (\ f\phi\ )\ |V_{f\phi}^{pre_V(a)}|)$$

  which reads: the value of the λ-fluent $f\phi$ should be *greater-than* the cardinality of $f\phi$ in $pre_V(a)$.
- **Add effects** involving the variables in V, $add_V(a)$, are converted in *increase* effects:

$$add_V(a) \text{ are replaced by } (increase\ (\ f\phi\ )\ |V_{f\phi}^{add_V(a)}|)$$

- **Delete effects** involving the variables in V, $del_V(a)$ are converted in *decrease* effects:

$$del_V(a) \text{ are replaced by } (decrease\ (\ f\phi\ )\ |V_{f\phi}^{del_V(a)}|)$$

```
(:action hold
    :parameters (?x - slice ?y - tray)
    :precondition (and (ontray ?x ?y) (freearms))
    :effect (and (not (ontray ?x ?y))
                 (holding ?x ?y) (not (freearms))))
```

```
(:action hold
    :parameters (?y - tray ?z - size)
    :precondition (and (freearms)
                       (>= (ontray_pizzasize_slice ?y ?z) 1))
    :effect (and (not (freearms))
                 (decrease (ontray_pizzasize_slice ?y ?z) 1)
                 (increase (holding_pizzasize_slice ?y ?z) 1)))
```

Figure 2. Example of original and compiled actions.

Figure 2 shows an example action of the pizza domain to illustrate Definitions 4 and 5, and the rewriting rules. The original action is at the top and the compiled action at the bottom.

Additionally, we remove from F_t those λ-fluents not unifying with the preconditions of any action, since they are not useful. An additional reachability analysis could be performed to also remove λ-fluents not reachable from the initial state: those having a value of zero at the initial state for which any action increases their value.

The translation of $\mathcal{I}$ into $\mathcal{I}'$ consists of including in the initial state the value of relevant λ-fluents while removing literals regarding $\mathcal{P}_t$. λ-fluents unifying with $\mathcal{I}$ have a value equal to their cardinality. The syntactic rule replaces matching literals with an *equal-to* assignment:

$$\bigcup_{v \in V_{f\phi}^{\mathcal{I}}} \mathcal{I}_v \text{ is replaced by } (= (f\phi)\ |V_{f\phi}^{\mathcal{I}}|)$$

The rest of λ-fluents (those not unifying with the initial state) have an initial value of zero.

After the translation, the task $\Pi_{\lambda(t)}$ can be solved with a planner supporting numeric state variables. The resulting plan for $\Pi_{\lambda(t)}$ encompasses the equivalence class of any permutation of symbols having the same object sub-state in $\mathcal{I}$.

Proposition 1. If a plan $\pi' = \langle a'_1, \ldots, a'_n \rangle$ solves the task $\Pi_{\lambda(t)}$, there exists a plan $\pi = \langle a_1, \ldots, a_n \rangle$ that solves Π_t.

Proof Sketch: Let $\langle s'_0, \ldots, s'_n \rangle$ be the state sequence induced by the application of π'. State $s'_0 = I'$ is well formed by the definition of the compilation. Actions a'_i without λ-fluents in $pre(a')$ are not modified in the compilation. If an action a'_i with some λ-fluents f in $pre(a')$ is applicable, there are sufficient objects in the state s_i of Π_t matching *atoms(f)*, therefore an action a_i is applicable in s_i. In fact, all matching atoms enable a set of actions which are permutations over the regarding objects. For reconstructing π from π' is enough to select one of these permutations for each a'_i. □

4.1 Object Creation Tasks

The compilation Λ is applicable to any task fulfilling the condition of irrelevancy of object names. Now, we distinguish tasks related to the creation of objects.

Definition 6. Task $\Pi_{\hat{t}}$ is an **object creation task** if an unary non-static predicate $\varepsilon(t) \in \mathcal{P}_t$ holds the following for all $a \in \mathcal{A}$:

- $\varepsilon(t) \notin add(a)$.
- if $\varepsilon(t) \in del(a)$, then $\exists p'(t) \in \mathcal{P}_t$ such that $p'(t) \in add(a)$ (i.e. predicates p' define the initial properties of the created object)
- if $\varepsilon(t) \in pre(a)$, then $\nexists p'(t) \in \mathcal{P}_t$ such that $p'(t) \in pre(a)$.
- and, if $p(o) \in \mathcal{I}$, $o \in \mathcal{C}_t$, then there is no other fact over o in $\mathcal{I}$ (i.e. they are free in the pool of objects of type t).

$\varepsilon(t)$ is the special predicate encoding that a symbol has not been used in the pool model[7]. In addition, $\varepsilon(t)$ belongs to the mutex group formed by the predicates stated as initial properties of the new objects. In $\Lambda(\Pi_{\hat{t}})$, $\varepsilon(t)$ will become a special λ-fluent ε_t that counts the number of free symbols in the initial state. When C_t is defined just to have enough symbols to solve $\Pi_{\hat{t}}$ rather than to limit the set of objects that can be created, the compilation $\Lambda(\Pi_{\hat{t}})$ can ignore λ-fluent ε_t to simulate an arbitrarily large C_t.

Definition 7. Given an object creation task $\Pi_{\hat{t}}$, the **compilation** $\Lambda^{\sim\varepsilon}(\Pi_{\hat{t}})$ generates $\Pi^{\sim\varepsilon}_{\lambda(t)}$, an approximate translation of $\Pi_{\hat{t}}$. $\Pi^{\sim\varepsilon}_{\lambda(t)}$ is $\Pi_{\lambda(t)}$ with all appearances of ε_t removed. That is, $\Pi^{\sim\varepsilon}_{\lambda(t)} = (\mathcal{T}', \mathcal{C}', \mathcal{P}', F_t \backslash \{\varepsilon\_t\}, \mathcal{A}'', \mathcal{I}', \mathcal{G})$, where every $a'' \in A''$ is the result of removing all preconditions and effects involving ε_t from $a' \in A'$.

Proposition 2. If a plan π that solves the object creation task $\Pi_{\hat{t}}$, exists after making $|C_t| > M$, there exists a plan π'' that solves $\Pi^{\sim\varepsilon}_{\lambda(t)}$ for an arbitrary M.

Proof Sketch: If we apply Λ instead of $\Lambda^{\sim\varepsilon}$, the plan π'' will exists iff $\varepsilon\_t > M$ holds in $\mathcal{I}'$, given that $\varepsilon(t)$ is not added, and therefore ε_t is not increased. Thus, if a plan π'' exists when $\varepsilon\_t > M$, it is because some action a' with $\{\varepsilon\_t > k\} \in pre(a')$ has become applicable for generating π''. Consider now the case for $\Lambda^{\sim\varepsilon}$. For each state s', an action a' from which ε_t has been removed (i.e., a is a *creating action*) is applicable if the remaining $pre(a')$ holds in s'. In these cases a' is applicable regardless the value of M. □

Plans π'' will also solve tasks with a number of objects in C_t smaller than M. Strictly speaking, these are invalid plans for $\Pi_{\hat{t}}$. However $\Lambda^{\sim\varepsilon}$ might be intentionally used when $|C_t|$ is not a constraint of the problem, such as a fixed availability of resources. The approximate translation is useful when the number of symbols of type t is irrelevant for modelling the task but needed to solve it.

5 IMPLEMENTING PLAN TRANSLATION

The key issue for the translation of a plan from the numerical to the pool model is to give names to the objects counted in the numeric model that should appear as action parameters in the plan for the pool model. This is handled by associating a stack of symbols to each λ-fluent, including ε_t. Initially, each of these stacks contains the constants in the initial state of the pool model whose atoms unify with the corresponding λ-fluent of the numerical model. Since λ-fluents define complete sub-states, effects of actions in the numerical model serve to move symbols from a stack to another. Thus, the execution of the numeric plan is simulated as a state machine. Decrease effects indicate the stack where symbols are removed from, while increase effects indicate the stack in which those symbols are included. Moved symbols are the parameters of the corresponding action in the plan for the pool model. If there are only decrease effects in an action, moved objects are not placed in any stack (i.e. the objects disappear).

[7] An alternative, but more elaborated definition of object creation tasks is also feasible for the control model.

The ε stack has a specific behavior. When it becomes empty and some symbols need to be extracted from it, a procedure automatically generates new symbols not declared in the pool model. Thus, the number of needed symbols is decided after planning and not before.

6 EXPERIMENTS

In this section, we evaluate the performance of the *Numeric Resource Creator Planner* (NUMRECREP). This planner receives as inputs the domain and problem in the pool model, performs the compilation $\Lambda^{\sim\varepsilon}(\Pi_{\hat{t}})$, solve the problem in the numeric model and finally compiles the solution plan back to the pool model.

Traditional planning benchmarks do not contain creation actions, not even created resources with changing properties. Therefore, we have developed four new domains with these characteristics for objects of one type. We first describe them and then we report some results of the compilation effectiveness.

Carpenter: a modified version of the Woodworking domain of previous International Planning Competitions (IPCs). Now wood "parts" do not exist initially, so they have to be created sawing a board. From a board, one can create small parts, and large parts. As the original domain, color may be added (glaze) or removed (plane) from a part. Goals consist of building stools and benches, which are composed by different parts with a particular color and size.

Pizza: a robot waiter splits pizzas and serve slices of the same size. The domain for the running examples during the paper.

Child-Snack: to make and serve sandwiches for a group of children in which some of them are allergic to gluten. There are two actions creating objects. The first one makes a sandwich and the second one makes a sandwich taking into account that all ingredients are gluten-free. This domain is useful to show that we are able to handle created objects with different number of properties.

Spanner-prime: a slight variation of the domain of the IPC-2011 Learning Track. In the original version a man has to pick up a set of spanners placed along a path to tighten a set of nuts. In the *spanner'* domain, spanners do not exists initially, but the man can obtain a new created spanner dispensed in any location of the path.

Setup For each domain we developed a random problem generator able to pre-compute the number of needed symbols for non-existing objects. It generates a number of symbols according to a ratio given as input parameter. A ratio of 100% guarantees solvability for classical models (i.e., pool and control). This ratio is irrelevant for NUMRECREP, since it creates symbols on demand.

For the evaluation of the classical models we used the LAMA planner [11], winner of the Satisficing Track of IPC-11. LAMA does not support numeric preconditions, so we selected Metric-FF [8] as the base solver for NUMRECREP. We will report also the results of Metric-FF with the classical model, to establish a fair comparison of different models with the same planner. The time bound for solving a problem was set to 1800 seconds, as in last IPCs. Experiments were run in a 2.4Ghz CPU with 6GB of memory bound.

Object Increase In this experiment we analyze the effect of declaring a different number of initial symbols for the object type being created. For each domain, we have generated a set of 20 random problems with the same distribution of objects and a fixed set of goals. The number of declared symbols in these problems varies from 110% up to 300% of the needed symbols to solve the problems. Then, we sorted these problems by their size. It is expected that classical planners with larger problems will suffer from scalability prob-

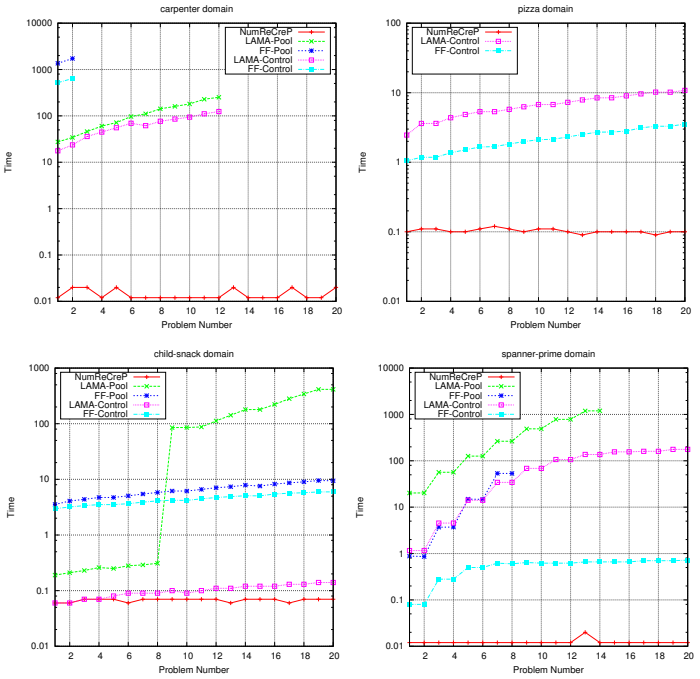

Figure 3. Performance for fixed goals and increasing number of constants in the same problem.

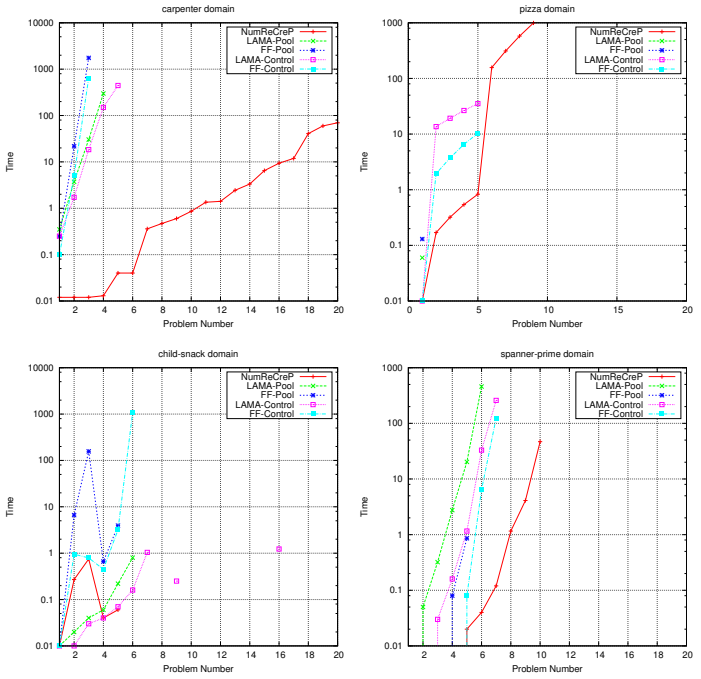

Figure 4. Performance for problems with increasing number of goals.

lems due to the increase in the instantiation and state space. However, in the numeric model no symbols are declared for the type being created. Thus, in this model the translated problems are all of a similar size and the differences on performance will be due only to CPU latency. Figure 3 shows the results in terms of CPU time. As expected, planners perform better with control models than with pool models. NUMRECREP obtains an impressive improvement compared to other planners. Even the control model degrades the performance of planners when more symbols than needed are defined.

Scalability The objective of this experiment is to analyze how planners with different models scale when we increase the size of the planning instances. For each domain, we have generated a set of 20 random problems with increasing number of goals. The sets of objects were increased proportionally, and the number of symbols for the object type being created was set to 120% of the needed objects. Figure 4 shows the results in terms of CPU time. NUMRECREP scales better than the others in three out of four domains. In all cases NUMRECREP correctly solves the issue of the hard instantiation. The difference among domains is due to the ability of the Metric-FF heuristic to guide the search. Problems in Carpenter are hard to instantiate, but the relaxed plan heuristic performs reasonably well regardless of the model. The Child-snack domain is also hard to instantiate but landmarks heuristic is more informative than Metric-FF heuristic. Regarding the pizza domain, the largest problem solved by LAMA in the control model has 2 pizzas and 8 guests. NUMRECREP was able to solve problems with 3 pizzas and 12 guests. To some extend it is surprising that such an easy task can not be solved by state-of-the-art planners.

7 CONCLUSIONS

The creation of objects is a common situation in planning scenarios where agents need to assembly products, split objects, or even assign names to objects being perceived from the environment. In this work we have identified under which conditions these objects can be compiled as numeric state variables. Then, we have proposed a new method to compile them automatically. This approach reliefs the problem of explicitly declaring a sufficient number of non-existing object names in the problem definition. The translation into numeric models has also the advantage of reducing the instantiation and removing symmetries due to irrelevant object names.

As future work we want to incorporate a reachability analysis for pruning non-reachable λ-fluents and to study other mechanisms to reduce its number. Also, we want to empirically explore the benefits of the translation when dealing with resources that already exist in the problem definition but their names are irrelevant as well.

References

[1] Blai Bonet, Patrick Haslum, Sarah Hickmott, and Sylvie Thiébaux, 'Planning via Petri net unfolding: generalisation and improvements', in *Proceedings of ICAPS*, (2007).

[2] Maria Fox and Derek Long, 'The automatic inference of state invariants in TIM', *Journal of Artificial Intelligence Research (JAIR)*, **9**, 317–371, (1998).

[3] Maria Fox and Derek Long, 'The detection and exploitation of symmetry in planning problems', in *Proceedings of the IJCAI*, (1999).

[4] Hector Geffner, 'Functional STRIPS: a more flexible language for planning and problem solving', in *Logic-Based Artificial Intelligence*, Kluwer, (2000).

[5] Patrik Haslum and Ulrich Scholz, 'Domain knowledge in planning: Representation and use', in *Proceedings of the ICAPS 2003 Workshop on PDDL*, (2003).

[6] Malte Helmert, 'Concise finite-domain representations for PDDL planning tasks', *Artificial Intelligence (AI)*, **173**, 503–535, (2009).
[7] Sarah Hickmott, Jussi Rintanen, Sylvie Thiébaux, and Lang White, 'Planning via Petri net unfolding', in *Proceedings of the IJCAI*, (2007).
[8] Jörg Hoffmann, 'The METRIC-FF planning system: Translating "ignoring delete lists" to numeric state variables', *Journal of Artificial Intelligence Research (JAIR)*, **20**, 291–341, (2003).
[9] Bernhard Nebel, Yannis Dimopoulos, and Jana Koehler, 'Ignoring irrelevant facts and operators in plan generation', in *Proceedings of the ECP*, pp. 338–350, (1997).
[10] Nir Pochter, Aviv Zohar, and Jeffrey Rosenschein, 'Exploiting problem symmetries in state-based planners.', in *Proceedings of the AAAI*, (2011).
[11] Silvia Richter and Matthias Westphal, 'The LAMA planner: Guiding cost-based anytime planning with landmarks', *Journal of Artificial Intelligence Research (JAIR)*, **39**, 127–177, (2010).